Tutoriel sur gRPC

IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

1. Objectif

L'objectif de ce tutoriel est de vous introduire les concepts de base pour utiliser gRPC. Nous allons pour cela travailler sur le service `Movie` précédemment codé par une interface REST/OpenAPI.

NOTE Pour tout détails sur gRPC suivez ce lien https://grpc.io/

IMPORTANT Tout comme pour `Flask` je considère ici que vous avez une version récente de `Python`, donc une version 3.x. J'utilise les commandes `python3` et `pip3` ayant sur ma machine plusieurs versions de `Python`. Vous pouvez vérifier votre version par défaut de `Python` avec `python --version`.

2. Installation de gRPC Python

Activez votre environnement virtual Python pour le TP :

```
source <venv>/bin/activate
```

NOTE voir ici pour plus de détails https://docs.python.org/3/library/venv.html

Nous pouvons maintenant installer gRPC Python dans l'environnement virtuel.

```
pip3 install grpcio
pip3 install grpcio-tools # to handle protocol buffers
```

NOTE pour plus de détails voir https://grpc.io/docs/languages/python/quickstart/

3. Vérifier l'installation

```
git clone -b v1.40.0 https://github.com/grpc/grpc
cd grpc/examples/python/helloworld
```

Et dans deux terminaux distincts :

```
python3 greeter_server.py
```

```
python3 greeter_client.py
```

4. Protocol buffers et génération des stubs

Commencez par créer deux répertoires, l'un pour le service `Movie` et l'autre pour le `Client` de ce service.

4.1. Création d'une première interface avec Protocol Buffers

Voici l'interface de base Protocol Buffers que nous allons utiliser. Vous pouvez l'enregistrer dans vos deux répertoires de travail `movie/protos/movie.proto` et `client/protos/movie.proto`.

```
syntax = "proto3";

service Movie {
    rpc GetMovieByID(MovieID) returns (MovieData) {}
}

message MovieID {
    string id = 1;
}

message MovieData {
    string title = 1;
    float rating = 2;
    string director = 3;
    string id = 4;
}
```

Ce fichier représente le **contrat** entre le serveur et le client sur l'API à utiliser.

Nous y indiquons la version de Protocol Buffers utilisée, ici `proto3`.

Nous définissons ensuite notre service `Movie` qui va contenir une seule procédure distante, identifiée par le mot `rpc` : `GetMovieByID`. Cette prcédure ou méthode prend en entrée un message de type `MovieID` et retourne un message de type `MovieData`.

Enfin nous définissons les types de messages utilisés par les procédures `rpc`. Ici `MovieID` est un message contenant simplement un identifiant de type `string`. Et `MovieData` est un message contenant les éléments qui constituent un film dans notre fichier `JSON` : un titre de type `string`, une note de type `float`, un directeur de typ `string`, et enfin l'identifiant du film de type `string` comme pour `MovieID`. Les valeurs entières données à chaque élément du message servent au compilateur

proto pour identifier et ordonner les éléments constituants le message.

4.2. Compilation du fichier movie.proto

Pour compiler ce fichier nous devons lancer la commande suivante :

```
python3 -m grpc_tools.protoc -I=./protos --python_out=. --grpc_python_out=.
movie.proto
```

- -I= précise le répertoire source qui contient les fichiers proto
- --python_out= and --grpc_python_out= précise le répertoire de destination des fichiers Python générés
- enfin, le fichier proto à compiler est indiqué

NOTE	Pour plus de détail sur le langage Protocol Buffers voir https://developers.google.com/protocol-buffers/docs/pythontutorial

4.3. Analyse des fichiers générés par la compilation

Deux fichiers ont été générés suite à cette compilation du fichier movie.proto

- movie_pb2_grpc.py : ce fichier contient les éléments relatifs aux stub
- movie_pb2.py : ce fichier contient le code relatif aux types de messages et le marshalling et unmarshalling des informations (nous ne regarderons pas plus en détail ce fichier)

Avec gRPC le stub du client est appelé le stub alors que le stub du serveur est appelé le servicer.

4.3.1. stub client

Une classe est créée pour le stub. Ici elle se nomme MovieStub et notre client devra hériter de cette classe. Elle contient un constructeur qui prend en entrée un channel gRPC pour créer ses connexions avec le serveur. Pour chaque méthode/procédure du service précisé dans le fichier proto initial, un attribut portant le même nom est ajouté à l'objet MovieStub. Ce sont ces attributs que notre client devra appeler pour effectuer des appels distants au service. L'attribut créée une connexion avec le serveur en utilisant l'input channel (4 types possibles unaryunary, unarystream, streamunary et streamstream).

4.3.2. servicer serveur

Pour chaque service, une classe servicer est créée (ici une seule nommée MovieServicer). Pour chaque rpc indiqué dans le fichier proto initial une méthode est ajoutée dans la classe MovieServicer. Cette méthode devra être surchargée dans l'implémentation du service.

4.3.3. fonction d'enregistrement

Pour chaque service (ici un seul) une fonction pour enregistrer un `servicer` qui implémente le serveur est générée. Ici cette fonction est `add_MovieServicer_to_server`.

5. Exemple d'écriture du service `Movie`

Nous créons un fichier `movie/movie.py`.

5.1. Création et lecture des données JSON

Tout comme pour le TP sur REST nous allons utiliser le fichier `movie.json` disponible sur Moodle et le placer dans un répertoire `databases` (ou un autre nom de votre choix).

Le code ci-dessous permet la lecture d'un fichier JSON. L'objet `movies` récupéré est obtenu en lisant la clé "movies" et est donc un `array`.

```
import json

with open('{}/databases/movies.json'.format("."), "r") as jsf:
   movies = json.load(jsf)["movies"]
```

5.2. Implémentation du `servicer`

Nous allons maintenant devoir fournir une implémentation de l'objet `MovieServicer` généré.

Tout d'abord nous devons importer les deux fichiers générés `movie_pb2` et `movie_pb2_grpc`.

Nous créons ensuite une classe qui hérite de l'objet `movie_pb2_grpc.MovieServicer` généré. Dans cette classe nous surchargeons le constructeur où nous lisons le fichier `json` qui nous servira de base de données.

```
import json
import grpc
from concurrent import futures
import movie_pb2
import movie_pb2_grpc

class MovieServicer(movie_pb2_grpc.MovieServicer):

    def __init__(self):
        with open('{}/databases/movies.json'.format("."), "r") as jsf:
            self.db = json.load(jsf)["movies"]
```

Nous implémentons ensuite la méthode `GetMovieByID`. Cette fonction parcourt les films de la base de données. Si l'identifiant fournit dans la requête est trouvé il est retourné par le serveur dans sa

réponse à l'appel de procédure distant. Pour cela le descripteur de fichier `movie_pb2.MovieData` est utilisé pour construire le message à retourner.

```
class MovieServicer(movie_pb2_grpc.MovieServicer):

   ...

    def GetMovieByID(self, request, context):
        for movie in self.db:
            if movie['id'] == request.id:
                print("Movie found!")
                return movie_pb2.MovieData(title=movie['title'], rating=movie['rating
'], director=movie['director'], id=movie['id'])
        return movie_pb2.MovieData(title="", rating="", director="", id="")
```

Enfin nous devons créer une fonction `main` qui va se charger de créer le serveur `gRPC` et enregistrer la classe `MovieServicer` à ce serveur. Nous ouvrons ici le port `3001`.

```
def serve():
    server = grpc.server(futures.ThreadPoolExecutor(max_workers=10))
    movie_pb2_grpc.add_MovieServicer_to_server(MovieServicer(), server)
    server.add_insecure_port('[::]:3001')
    server.start()
    server.wait_for_termination()


if __name__ == '__main__':
    serve()
```

6. Exemple d'écriture du client

Pour implémenter le client il n'y a pas de classe à implémenter, il faut en revanche initier le `stub` en créant une connexion avec le serveur et faire appel au `stub` lors de l'appel de procédures distantes.

Nous créons un fichier `client/client.py`.

Tout d'abord nous devons importer les éléments nécessaires pour coder le client, à savoir `grpc`, ainsi que les deux fichiers générés par la compilation.

```
import grpc
import movie_pb2
import movie_pb2_grpc
```

Ici nous commencons par une fonction `main`. Nous créons une connexion avec l'hôte `localhost:3001` et utilisons ce channel pour créer le `stub movie_pb2_grpc.MovieStub(channel)`. Puis nous appelons la fonction interne `get_movie_by_id`.

Cette fonction prend en paramètres le `stub` et le message d'entrée de l'appel distant que nous créons au moyen du descripteur `movie_pb2.MovieID`.

```
def run():
    with grpc.insecure_channel('localhost:3001') as channel:
        stub = movie_pb2_grpc.MovieStub(channel)

        print("-------------- GetMovieByID --------------")
        movieid = movie_pb2.MovieID(id = "a8034f44-aee4-44cf-b32c-74cf452aaaae")
        get_movie_by_id(stub, movieid)


if __name__ == '__main__':
    run()
```

NOTE Pour en savoir plus sur les channels en `gRPC` voir la documentation https://grpc.github.io/grpc/python/grpc.html

Nous allons maintenant implémenter la fonction `get_movie_by_id`. Elle va tout simplement appeler la fonction du `stub GetMovieByID` en utilisant le message donné en entrée de type `movie_pb2.MovieID`. Le message de réponse est stocké dans la variable `movie` et est affichée.

```
def get_movie_by_id(stub,id):
    movie = stub.GetMovieByID(id)
    print(movie)
```

7. Exécution de nos services

Nous lancons le serveur tout d'abord.

```
cd path/movie
python3 movie.py
```

Puis le client.

```
cd path/client
python3 client.py
```

Nous devrions observer cette sortie côté client, c'est à dire les informations relatives au film ayant l'identifiant `a8034f44-aee4-44cf-b32c-74cf452aaaae` en entrée de l'appel de procédure distante :

```
-------------- GetMovieByID --------------
title: "The Martian"
rating: 8.199999809265137
director: "Ridley Scott"
id: "a8034f44-aee4-44cf-b32c-74cf452aaaae"
```

Et cette sortie côté serveur :

```
Movie found!
```

8. Modification de l'API

Nous allons maintenant ajouter une fonction à notre API RPC. Voici le nouveau fichier `movie.proto` qui est bien entendu à modifier côté serveur et côté client puisqu'il représente le contrat de l'API.

```
syntax = "proto3";

service Movie {
    rpc GetMovieByID(MovieID) returns (MovieData) {}
    rpc GetListMovies(Empty) returns (stream MovieData) {}
}

message MovieID {
    string id = 1;
}

message MovieData {
    string title = 1;
    float rating = 2;
    string director = 3;
    string id = 4;
}

message Empty {
}
```

Nous ajoutons donc une procédure distante `GetListMovies` qui prend un message d'entrée vide de type `Empty` et qui retourne un `stream` de messages de type `MovieData`.

Nous allons regénérer les fichiers `movie_pb2` et `movie_pb2_grpc` permettant d'abstraire l'appel de procédure distant de le code de notre serveur et notre client.

TIP Vous pouvez soit exécuter la commande des 2 côtés, soit copier/coller les fichiers générés côté serveur et côté client.

```
python3 -m grpc_tools.protoc -I=./protos --python_out=. --grpc_python_out=.
movie.proto
```

Jetez un coup d'oeil aux nouveaux fichiers générés qui doivent inclure le nouvel appel RPC possible dans notre API !

Nous n'avons plus qu'à compléter notre serveur et notre client.

Voici le code de notre méthode `GetListMovies` dans la classe `MovieServicer` :

```
def GetListMovies(self, request, context):
    for movie in self.db:
        yield movie_pb2.MovieData(title=movie['title'], rating=movie['rating'],
director=movie['director'], id=movie['id'])
```

Nous parcourons les films de la base de données, nous créons un message par film de type `MovieData` et au lieu de retourner notre message nous utilisons le mot clé `yield` qui va permettre à `gRPC` de mettre en place le stream de messages.

Voici maintenant le code de notre client auquel nous avons ajouté une fonction interne `get_list_movies` ne prenant comme entrée que le `stub`.

```
def get_list_movies(stub):
    allmovies = stub.GetListMovies(movie_pb2.Empty())
    for movie in allmovies:
        print("Movie called %s" % (movie.title))
```

Dans cette fonction nous faisons l'appel de procédure distante en utilisant le `stub` initialisé et en créant un message vide de type `Empty` (qui a été ajouté au fichier `movie_pb2`). Le stream de messages est aggégé dans la variable `allmovies`. Nous parcourons ces éléments et les affichons par titre.

Nous devons évidemment appeler cette fonction dans le `main` du client :

```
print("-------------- GetListMovies --------------")
get_list_movies(stub)
```

En redémarrant le serveur et le client vous devez observer cette nouvelle sortie côté client :

```
-------------- GetMovieByID --------------
title: "The Martian"
rating: 8.199999809265137
director: "Ridley Scott"
id: "a8034f44-aee4-44cf-b32c-74cf452aaaae"

-------------- GetListMovies --------------
Movie called The Good Dinosaur
Movie called The Night Before
Movie called Creed
Movie called Victor Frankenstein
Movie called The Danish Girl
Movie called Spectre
```