Distributed Systems Programming

A.Y. 2023/24

Laboratory 2

In this laboratory activity, you are invited to practice with gRPC, a modern open-source high performance framework implementing the remote procedure call (*RPC*) paradigm.

The activity consists in the implementation of a gRPC client (in node.js) and its integration within the *Film Manager* service developed in Laboratory 1. This client will have to interact with a Java gRPC server. The data structures (called messages) and services involved in the communication between the gRPC client and server will be described in a .proto file (Protocol Buffer file).

The tools that are recommended for the development of the solution are:

- *Visual Studio Code* (https://code.visualstudio.com/) for the extension of the *Film Manager* service by implementing a gRPC client;
- *vscode-proto3,* extension of Visual Studio Code, (https://marketplace.visualstudio.com/items?itemName=zxh404.vscode-proto3), for the definition of Protocol Buffer files;
- *Eclipse IDE for Enterprise Java Developers* for the development of the gRPC server;
- *PostMan* (https://www.postman.com/) for testing the extended version of the *Film Manager* service;
- *DB Browser for SQLite* (https://sqlitebrowser.org/) for the management of the database for the *Film Manager* service.

Context of the activity

The *Film Manager* service is extended with a new functionality with respect to the service version developed for Laboratory 1, i.e., it offers the possibility to associate a set of *images* to each public film (e.g., screenshots, posters, flyers). The following three image media types are supported by the service: PNG (.png), JPEG (.jpg), GIF (.gif). No other image media types are allowed by the service.

The *Film Manager* service exposes the following REST APIs, related to the association of images to films, only for authenticated users:

- A user can associate a new image to a public film she owns, by uploading the image file to the *Film Manager* service. Multiple images can be associated with a single film. When the service receives an image with an accepted image media type, it locally stores the image file with its media type, and saves the related information (i.e., name of the image, and association of the image to the film) as a data structure, named *image* for simplicity, in the database.
- A user can retrieve the list of all the images associated to a public film she owns, or she is a reviewer of. In this case, the service will return a json-encoded array of image data structures. A pagination mechanism is not necessary for their retrieval, as the typical use case addressed by the design is to associate a limited number of images to each film.
- A user can retrieve each single image associated to a public film she owns, or she is a reviewer of. In this case, by specifying the desired content type via the *Accept* header, the user can decide whether to retrieve the image data structure (json content type), which does not contain the image file, or the image file itself, in one of the supported image content types (image/png, image/jpg, and image/gif). In case the user requests another media type, the operation fails.
- A user can delete an image associated to a public film she owns (in this case, the service deletes the corresponding image file(s) as well).

When a user requests an image file with a certain media type and the *Film Manager* service has that image file already locally stored with that media type, it immediately sends back the image file to the user.

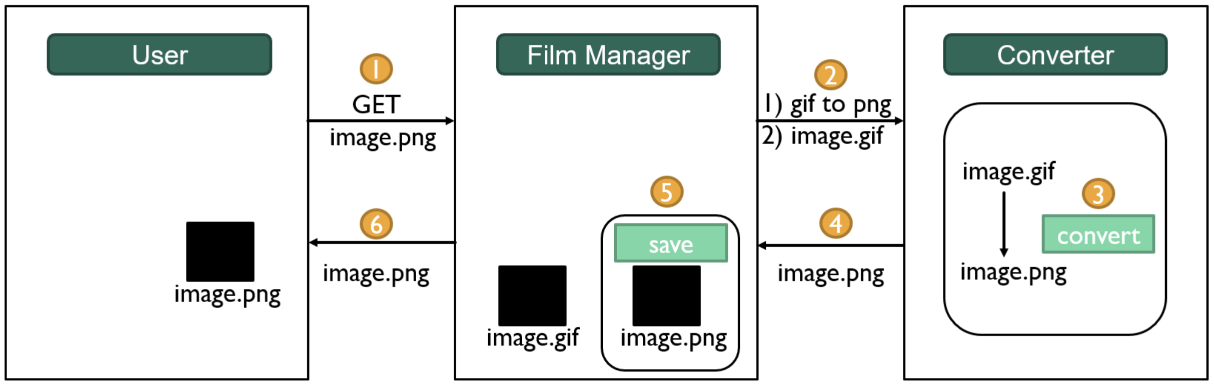
Instead, if the image file is locally available only with another media type, then the *Film Manager* service interacts with another service, called *Converter* service, delegating the operation of media type conversion. In greater detail, the *Film Manager* service uses a gRPC channel towards the *Converter* service (i.e., the *Film Manager* service covers the role of gRPC client, the *Converter* service covers the role of gRPC server). Their communication is organized in the following way:

- the *Film Manager* service sends a first messages describing the media type of the image to be converted, and the destination media type. Then, it sends the image file, divided into chunks.
- the *Converter* service converts the received image file into the requested media type, and sends it back to the *Film Manager* service, if no problem arises in the

conversion. Otherwise, it notifies why the conversion failed. The *Converter* service is stateless, i.e., it neither saves the received image file nor the result or the conversion.

After having received the chunks of the converted image file and having locally stored it, the *Film Manager* can finally send the file back to the user who requested it.

The following picture illustrates an example of a client requesting an image file (e.g., image.png) whose media type is not available server-side (e.g., only image.gif is available):



How to experience this laboratory activity

The full development of the extension involves the following tasks:

1. definition of the new REST APIs exposed by the *Film Manager* service;
2. definition of the .proto file on which the gRPC service is based;
3. extension of the node.js *Film Manager* service implementation, for the management of images (REST APIs implementation) and of the gRPC communication;
4. implementation of the gRPC *Converter* service in Java;

However, alongside this document, we provide you with the following items (already discussed in the course lectures):

- a .proto file specifying a possible organization of the messages and services involved in the gRPC *Converter* service;
- a full implementation of the Java-based gRPC server stub for the *Converter* service.

Hence, the Lab work consists of developing steps 1. and 3. However, you are strongly invited to have a careful look not only at the .proto file, which you need to use to implement the client side in step 3., but also at the code of the *Converter* service, and to understand both. The reason is that in the final exam you may be asked to extend them.

Useful tips

Here are some recommendations provided with the aim to help you in completing the tasks required by this Lab session activity:

- you can use the node.js *multer* (https://www.npmjs.com/package/multer) module to handle the reception and saving of image files in the *Film Manager* service. This module provides a middleware for handling *multipart/form-data*, which is the main way used for uploading files;
- you can use the node.js *@grpc/grpc-js* module (https://www.npmjs.com/package/@grpc/grpc-js) for implementing the functionality of gRPC client in the *Film Manager* service;
- you can use the node.js *@grpc/proto-loader* module (https://www.npmjs.com/package/@grpc/proto-loader) for loading .proto files to use with gRPC.

Here is some useful information related to the Java gRPC server:

- The .proto files must be put in the /src/main/proto folder of the Maven project.
- In order to automatically generate the required classes from the .proto file, in Eclipse you must synchronize the configuration of the project with the setting of the pom.xml. To do so, follow these instructions: Right click on the Eclipse project name → Maven → Update Project… → OK.
- The pom.xml file includes all the dependencies that are needed to use gRPC in Java.
- The compliance of the compiler must be set to 1.6 (or higher). The pom.xml file you can find together with the provided implementation of the server already includes the definition of the necessary properties to set the correct compliance level of the compiler to 1.11, which is higher than the minimum required one.