Programación Distribuida y Tiempo Real

Andrés Milla

Número de alumno: 14934/6
Email: andressmilla@gmail.com

1. Introducción

El informe está organizado de la siguiente manera: En la sección 2 se describe a Maven y su integración con gRPC mediante una guía paso a paso. Luego, en la sección 3 se describe la implementación de un servidor FTP en Java. Por último, en la sección 4 se muestra como utilizar Python para hacer uso de gRPC.

A continuación, se explica como está organizado el repositorio de código y se describe como instalar el entorno de ejecución.

1.1. Organización del repositorio de código

En el directorio `src` se encuentra el código fuente de todos los ejemplos utilizados, dentro de él hay cuatro implementaciones:

1. `java/greeter`: Es el ejemplo utilizado en la guía de Maven (Ver sección 2).
2. `java/ftp`: Contiene el servidor FTP desarrollado en Java que se utiliza en la sección 3.
3. `python/greeter`: Es un ejemplo adicional que muestra a Python como lenguaje para utilizar gRPC (Ver sección 4).
4. `python/ftp`: Es el servidor FTP desarrollado en Python (Ver sección 4).

1.2. Instalación del entorno de ejecución

Para instalar el entorno de ejecución se deberá contar con Docker [1]. Luego, **posicionados en la ubicación donde se encuentra este informe** se deberán ejecutar los siguientes comandos:

```
# Descargar la imagen de Docker
# Se monta el directorio "src" en la ubicación "/pdytr/"
SRC=$(pwd)/src

docker run -itd \
    -v $SRC:/pdytr/ \
    -p 5901:5901 -p 6901:6901 \
    --name pdytr gmaron/pdytr

# Instalar las dependencias
docker exec --user root -it pdytr bash /pdytr/install.sh
```

[1] https://www.docker.com/

1.3. Ejecución

Cada ejemplo utilizado tiene asociado un *Makefile* que permitirá ejecutar cada implementación de forma sencilla. Para ver el funcionamiento, **fuera del container** se deben ejecutar los siguientes comandos:

1. Implementación del servicio `Greeter` en Java:
   `docker exec --user root -it pdytr make -C /pdytr/java greeter`
2. Implementación del servicio `FTP` en Java:
   `docker exec --user root -it pdytr make -C /pdytr/java ftp`
3. Implementación del servicio `Greeter` en Python:
   `docker exec --user root -it pdytr make -C /pdytr/python greeter`
4. Implementación del servicio `FTP` en Python:
   `docker exec --user root -it pdytr make -C /pdytr/python ftp`

De todas formas, en cada sección se explica como ejecutar la implementación utilizada (por si se quiere ejecutar manualmente).

2. Maven

Maven es una herramienta utilizada para construir y administrar proyectos en Java. Principalmente, nos permitirá manejar **acciones ajenas** al código fuente, tales como el manejo de dependencias, la construcción del proyecto, distribución de nuestro código, entre otras. Dichas acciones, se especifican de forma declarativa utilizando el lenguaje XML a partir de un archivo llamado `pom.xml` [3].

2.1. Estructura mínima

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
            http://maven.apache.org/maven-v4_0_0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>pdytr.example.grpc</groupId>
    <artifactId>grpc-hello-server</artifactId>
    <version>1.0-SNAPSHOT</version>
</project>
```

Figura 1: Estructura mínima de un proyecto Maven

En la Fig. 1 se puede ver la estructura mínima de un proyecto Maven, la cual se compone por las siguientes etiquetas:

1. `<project>` (línea 1): Es la etiqueta raíz, a través de ella especificamos que utilizaremos el espacio de nombre de Maven (esto es propio de XML).

2. `<modelVersion>` (línea 5): Indica la versión del POM, en este caso se utiliza la última versión (4.0.0) compatible con la última versión de Maven.

3. `<groupId>` (línea 7): Indica el nombre del grupo del proyecto. En base a este nombre, Maven reemplazará los puntos con los separadores del Sistema Operativo que estemos utilizando. Por ejemplo, si utilizamos Linux, el grupo `pdytr.example.grpc` será convertido a `pdytr/example/grpc`. Luego de la conversión, todo archivo que generemos será guardado en esa ubicación.

4. `<artifactId>` (línea 8): Indica el nombre del proyecto, junto con el grupo permitirá armar la estructura de nuestro proyecto. En este caso, nuestro proyecto se encontrará en la siguiente ubicación `grpc-hello-server/pdytr/example/grpc`.

5. `<version>` (línea 9): Tal como lo indica su nombre, indica la versión de nuestro proyecto.

Cabe destacar que la unión de las últimas 3 etiquetas (`groupId:artifactId:version`) conforman el identificador que utiliza Maven para gestionar nuestros proyectos [4].

2.2. Integración con gRPC

A continuación se presenta una guía que permitirá crear, compilar y ejecutar nuestro proyecto Maven junto con la integración de gRPC. Como requisito se necesitará tener instalado el entorno de ejecución que se describe en la sección 1.2.

Para comenzar, deberemos ingresar a nuestro container de Docker mediante el siguiente comando:

```
docker exec --user root -it pdytr bash
```

Una vez dentro del container, creamos nuestro proyecto Maven mediante el siguiente comando:

```
mvn archetype:generate \
    -DgroupId=pdytr.example.grpc \
    -DartifactId=grpc-hello-server \
    -DarchetypeArtifactId=maven-archetype-quickstart \
    -DinteractiveMode=false
```

El comando `mvn` nos permite ejecutar las operaciones que provee Maven, en este caso la operación `archetype:generate` crea un nuevo proyecto utilizando una plantilla ya existente (Maven las denomina `archetype`). Luego, se especifican los siguientes argumentos:

1. `-DgroupId=pdytr.example.grpc`: Se especifica el nombre `pdytr.example.grpc` como *group id*.

2. `-DartifactId=grpc-hello-server`: Se especifica el nombre del proyecto. Tal como se explicó en la sección 2.1, el directorio raíz se llamará grpc-hello-server.

3. `-DarchetypeArtifactId=maven-archetype-quickstart`: Indica la plantilla (*archetype*) que se utilizará para generar el proyecto, en este caso se utiliza la recomendada por Maven para un inicio rápido.

4. `-DinteractiveMode=false`: Permite desactivar el modo interactivo, esto resulta de utilidad para crear de forma más rápida el proyecto.

Una vez ejecutado el comando, se generará la estructura de archivos que se puede observar en la Fig. 2.

```
grpc-hello-server/
|-- pom.xml
`-- src
    |-- main
    |   `-- java
    |       `-- pdytr
    |           `-- example
    |               `-- grpc
    |                   `-- App.java
    `-- test
        `-- java
            `-- pdytr
                `-- example
                    `-- grpc
                        `-- AppTest.java
```

Figura 2: Estructura de archivos generada al crear el proyecto Maven.

En la Fig. 2 podemos observar que se creó un directorio llamado *grpc-hello-server* con el siguiente contenido:

1. **Archivo** `pom.xml`: Como podemos observar en la Fig. 3, el archivo posee el `groupId` y el `artifactId` que fueron definidos en los argumentos del comando. También, se puede notar que como versión inicial, Maven nos asignó automáticamente el valor `"1.0-SNAPSHOT"`.
   Tal como sus nombres lo indican, las etiquetas `<name/>` y `<url/>` corresponden al nombre y URL del proyecto que se desean utilizar en la documentación. Mientras que la etiqueta `<packaging/>` indica que el proyecto se empaquetará en formato `.jar`.

Por último, la etiqueta `<dependencies/>` indica las dependencias que posee el proyecto, en este caso, la plantilla especificó al paquete `junit`. Además, en cada una de ellas se especifica como mínimo su identificador, es decir, las etiquetas `groupId`, `artifactId` y `version`.

Cabe destacar, que si en algún momento queremos cambiar la versión de alguna dependencia, simplemente tendremos que modificar el valor que se encuentra dentro de la etiqueta `<version/>`.

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
        http://maven.apache.org/maven-v4_0_0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>pdytr.example.grpc</groupId>
    <artifactId>grpc-hello-server</artifactId>
    <packaging>jar</packaging>
    <version>1.0-SNAPSHOT</version>
    <name>grpc-hello-server</name>
    <url>http://maven.apache.org</url>

    <dependencies>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>3.8.1</version>
            <scope>test</scope>
        </dependency>
    </dependencies>
</project>
```

Figura 3: Contenido del archivo pom.xml generado al crear el proyecto de Maven.

2. **Directorio** `src`: Aquí será en dónde escribiremos nuestro código fuente.

3. **Directorio** `test`: Tal como dice su nombre, aquí se colocarán los tests de unidad en caso de ser requeridos.

Como siguiente paso, deberemos añadir las librerías que permiten interactuar con gRPC. Para ello, deberemos editar el archivo `pom.xml` y agregar las dependencias que se observan en la Fig. 4.

```
<dependencies>
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>3.8.1</version>
        <scope>test</scope>
    </dependency>

    <!-- Dependencies for gRPC -->
    <dependency>
        <groupId>io.grpc</groupId>
        <artifactId>grpc-netty</artifactId>
        <version>1.7.0</version>
    </dependency>

    <dependency>
        <groupId>io.grpc</groupId>
        <artifactId>grpc-protobuf</artifactId>
        <version>1.7.0</version>
    </dependency>

    <dependency>
        <groupId>io.grpc</groupId>
        <artifactId>grpc-stub</artifactId>
        <version>1.7.0</version>
    </dependency>

    <dependency>
        <groupId>javax.annotation</groupId>
        <artifactId>javax.annotation-api</artifactId>
        <version>1.2</version>
    </dependency>
</dependencies>
```

Figura 4: Dependencias para integrar a gRPC al proyecto Maven.

Luego, deberemos especificar la etapa de `build` del proyecto. Esto nos servirá para generar automáticamente el código Java correspondiente a la especificación del archivo `proto`. En otras palabras, al ejecutar cualquier programa dentro de nuestro proyecto se generarán las clases especificadas en el archivo `proto`, sin necesidad de ningún comando adicional. Para ello, deberemos editar el archivo `pom.xml` y añadir el contenido de la Fig. 5 por debajo de la declaración de dependencias.

```
...
</dependencies>

<build>
  <extensions>
    <extension>
      <groupId>kr.motd.maven</groupId>
      <artifactId>os-maven-plugin</artifactId>
      <version>1.5.0.Final</version>
    </extension>
  </extensions>

  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>3.8.0</version>
      <configuration>
        <source>1.7</source>
        <target>1.7</target>
      </configuration>
    </plugin>

    <plugin>
      <groupId>org.xolstice.maven.plugins</groupId>
      <artifactId>protobuf-maven-plugin</artifactId>
      <version>0.5.0</version>
      <configuration>
        <protocArtifact>
          com.google.protobuf:protoc:3.4.0:exe:${os.detected.classifier}
        </protocArtifact>
        <pluginId>grpc-java</pluginId>
        <pluginArtifact>
          io.grpc:protoc-gen-grpc-java:1.7.0:exe:${os.detected.classifier}
        </pluginArtifact>
      </configuration>
      <executions>
        <execution>
          <goals>
            <goal>compile</goal>
            <goal>compile-custom</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>
```

Figura 5: Etapa de build del proyecto Maven.

Una vez especificadas las dependencias y la etapa de `build`, estamos en condiciones para escribir programas sencillos utilizando gRPC. A continuación, a modo de ejemplo se describirá como implementar un servicio llamado `Greeter`, el cual contará con una operación remota llamada `sayHello` que recibirá el nombre del cliente como argumento y le responderá con un mensaje de bienvenida.

Proto Deberemos crear un archivo `proto` que nos servirá para especificar las operaciones remotas del servicio. Luego, esta especificación será convertida a código Java, tal como se explicó en la etapa de `build`.

Nuestro código fuente debe estar la ubicación `grpc-hello-server/src/`. Por lo tanto, a partir de allí deberemos:

1. Crear un directorio llamado `proto` mediante el siguiente comando:
   `mkdir grpc-hello-server/src/main/proto`
2. Crear el archivo `greeter.proto` en el directorio creado anteriormente con el contenido de la Fig. 6. La ubicación del archivo debe ser `grpc-hello-server/src/main/proto/greeter.proto`.

```
1  syntax = "proto3";
2  package pdytr.example.grpc;
3
4  service Greeter {
5    rpc SayHello (HelloRequest) returns (HelloReply) {}
6  }
7
8  message HelloRequest {
9    string name = 1;
10 }
11
12 message HelloReply {
13   string message = 1;
14 }
```

Figura 6: Especificación **protobuff** del servicio `Greeter`

Tal como se puede observar en la Fig. 6, el archivo `proto` está compuesto por un servicio llamado *Greeter* (línea 4), con una operación remota denominada `SayHello`, la cual será implementada por el servidor y utilizada por el cliente. La misma, recibe como argumento un *message* `HelloRequest` (línea 8) que está conformado por el nombre del cliente (línea 9) y retorna otro *message* llamado `HelloReply` (línea 12) conformado por el mensaje de bienvenida que enviará el servidor (línea 13).
Al ejecutar nuestro proyecto, la etapa de `build` convertirá nuestra especificación *proto* en código Java, el cual va a estar disponible en el paquete `pdytr.example.grpc` (debido a la línea 2).

Servidor Nuestro servidor se encargará de atender las solicitudes de los clientes en el puerto 8080. Para ello, deberemos:

1. Editar el archivo `grpc-hello-server/src/main/java/pdytr/example/grpc/App.java` con el contenido de la Fig. 7, el cual servirá para exponer el servicio en el puerto 8080.

```
1  package pdytr.example.grpc;
2
3  import io.grpc.Server;
4  import io.grpc.ServerBuilder;
5
6  public class App {
7    public static void main(String[] args) throws Exception {
8      Server server = ServerBuilder
9          .forPort(8080)
10         .addService(new Greeter())
11         .build();
12
13     server.start();
14     System.out.println("Server started");
15
16     server.awaitTermination();
17   }
18 }
```

Figura 7: Implementación del servidor de gRPC.

Las líneas 8-11 configuran al servidor para escuchar en el puerto 8080 y atender las operaciones remotas mediante la clase `Greeter`, que será implementada en el paso siguiente. Luego, en la línea 13 se inicia el servidor en *background*, y por último en la línea 16 se espera a que la ejecución del servidor termine.

2. Crear el archivo `grpc-hello-server/src/main/java/pdytr/example/grpc/Greeter.java` con el contenido de la Fig. 8, en donde se encontrará el servicio `Greeter` junto con la implementación de la operación remota `sayHello`.

   En la Fig. 8 se puede notar lo siguiente:
   - En las líneas 5-6, los *messages* declarados en el archivo `proto` se generan en la clase `<nombre_del_servicio>OuterClass` (En nuestro caso, `GreeterOuterClass`) con sus respectivos *getters* y *setters*.
   - En la línea 7 se puede ver que la clase abstracta que contiene las operaciones remotas del servicio, está disponible en la clase `<nombre_del_servicio>Grpc` (En nuestro caso, `GreeterGrpc`) mediante una clase llamada `<nombre_del_servicio>ImplBase` (En nuestro caso, `GreeterImplBase`).

```java
package pdytr.example.grpc;

import io.grpc.stub.StreamObserver;

import pdytr.example.grpc.GreeterOuterClass.HelloRequest;
import pdytr.example.grpc.GreeterOuterClass.HelloReply;
import pdytr.example.grpc.GreeterGrpc.GreeterImplBase;

public class Greeter extends GreeterImplBase {
  @Override
  public void sayHello(HelloRequest request, StreamObserver<HelloReply> responseObserver) {
    String name = request.getName();
    System.out.println("Server received the name: " + name);

    String message = "Hello there, " + name;
    HelloReply response = HelloReply.newBuilder().setMessage(message).build();

    responseObserver.onNext(response);
    responseObserver.onCompleted();
  }
}
```

Figura 8: Implementación del servicio `Greeter`.

- En la línea 9 se implementa el servicio `Greeter` con la operación remota `sayHello`, que recibe como argumento el *message* `HelloRequest` (tal como fue declarado en el archivo `proto`) y un `observer`. Este último, nos servirá para responderle al cliente mediante el método `onNext` (línea 18) e indicar el cierre de conexión mediante el método `onCompleted` (línea 19). Se implementa de esta manera debido a que podemos tener diferentes tipos de direccionalidad en el intercambios de datos, en nuestro caso estamos utilizando un esquema *Unary*, es decir, el servidor envía un mensaje al cliente y se termina la interacción. En cambio, si enviáramos un mensaje más, estaríamos siguiendo un esquema *Server-streaming*. Para más información ver aquí [2].
- En la línea 16 se puede notar que para crear un *message* deberemos utilizar el método `newBuilder()`, luego establecer los valores a través de *setters* y finalmente crearlo con el método `build`.

[2] `https://grpc.io/docs/languages/java/generated-code/#unary`

Cliente Crearemos un cliente que se comunique con el servidor, y haga uso de la operación remota `sayHello`. Para ello, deberemos crear un nuevo archivo en la siguiente ubicación `grpc-hello-server/src/main/java/pdytr/example/grpc/Client.java` con el contenido de la Fig. 9

```
1  package pdytr.example.grpc;
2
3  import io.grpc.ManagedChannel;
4  import io.grpc.ManagedChannelBuilder;
5
6  import pdytr.example.grpc.GreeterGrpc;
7  import pdytr.example.grpc.GreeterGrpc.GreeterBlockingStub;
8  import pdytr.example.grpc.GreeterOuterClass.HelloRequest;
9  import pdytr.example.grpc.GreeterOuterClass.HelloReply;
10
11 public class Client {
12   public static void main(String[] args) throws Exception {
13     final ManagedChannel channel = ManagedChannelBuilder
14         .forTarget("localhost:8080")
15         .usePlaintext(true)
16         .build();
17
18     GreeterBlockingStub stub = GreeterGrpc.newBlockingStub(channel);
19     HelloRequest request = HelloRequest
20         .newBuilder()
21         .setName("Ray")
22         .build();
23
24     HelloReply response = stub.sayHello(request);
25     String greeting = response.getMessage();
26
27     System.out.println("Client received the greeting: " + greeting);
28
29     channel.shutdownNow();
30   }
31 }
```

Figura 9: Implementación del cliente de gRPC.

En la Fig. 9 podemos notar lo siguiente:

- Las líneas 13-16 configuran el canal que nos permitirá comunicarnos con el servidor a partir del puerto 8080.
- La línea 18 crea un *stub*. El mismo, contendrá los métodos con las operaciones remotas definidas en el servicio para poder interactuar con el servidor. Particularmente, se opta por un *stub* bloqueante, lo que significa que al llamar a cualquier operación remota, la ejecución permanecerá bloqueada hasta

obtener respuesta (Para más información ver aquí [3]). Notar que las utilidades para el cliente están definidas en la clase `<nombre_del_servicio>Grpc` (En nuestro caso, `GreeterGrpc`).
- La línea 24 realiza el llamado a la operación remota `sayHello` y posteriormente se extrae el mensaje de la respuesta para mostrarlo en pantalla (líneas 25-27).
- Por último, la línea 29 cierra el canal establecido previamente.

Pongamos en funcionamiento lo que hemos desarrollado. Para ejecutar el **servidor**, deberemos ir al directorio `grpc-hello-server` y ejecutar el siguiente comando:

```
mvn -DskipTests package \
    exec:java \
    -Dexec.mainClass=pdytr.example.grpc.App
```

Para ejecutar el **cliente**:

```
mvn -DskipTests package \
    exec:java \
    -Dexec.mainClass=pdytr.example.grpc.Client
```

Notemos que el ejecutable mvn recibe los siguientes parámetros:

1. `exec:java`: Permite ejecutar el proceso de Java especificado.
2. `-Dexec.mainClass`: Se indica la clase que se ejecutará, hay que tener en cuenta que debe estar acompañada con el paquete correspondiente.
3. `-DskipTests`: No ejecuta los tests que se han escrito en el directorio `test`. En caso de implementar tests, se deberá quitar este argumento.

La ejecución de los anteriores comandos producirá como salida el contenido que se aprecia en las figuras 10 y 11.

```
$ mvn -DskipTests package \
    exec:java \
    -Dexec.mainClass=pdytr.example.grpc.App

> Server started

> Server received the name: Ray
```

Figura 10: Salida al ejecutar el Servidor gRPC.

```
$ mvn -DskipTests package \
    exec:java \
    -Dexec.mainClass=pdytr.example.grpc.Client

> Client received the greeting: Hello there, Ray
```

Figura 11: Salida al ejecutar el Cliente gRPC.

Por último, cabe destacar los siguientes aspectos:

[3] `https://grpc.io/docs/languages/java/generated-code/#client-stubs`

1. En la primera ejecución, Maven deberá descargar todas las dependencias de nuestro proyecto, por lo cual tardará un tiempo considerable en ejecutar los programas. Sin embargo, en las siguientes ejecuciones permanecerán en la caché, por lo que no se necesitarán descargar de nuevo.

2. En caso de modificarse el archivo `proto`, Maven se encargará de generar el nuevo código automáticamente, no deberemos ejecutar ningún otro comando adicional.

3. Si se desea subir el proyecto Maven a algún repositorio de código, podemos utilizar el siguiente archivo .gitignore[4] para no subir los binarios generados por Maven.

4. En caso de querer eliminar los archivos generados por Maven, deberemos ejecutar el siguiente comando:

   ```
   mvn clean:clean
   ```

3. Caso de estudio: Servidor FTP

En esta sección se detalla la implementación de un servidor FTP, el cual es requerido por el inciso 4.a de la Práctica 3. El código fuente se encuentra en el directorio `src/java/ftp/`.

3.1. Enunciado

Con la finalidad de contar con una versión muy restringida de un sistema de archivos remoto, en el cual se puedan llevar a cabo las operaciones enunciadas informalmente como:

- **Leer**: Dado un nombre de archivo, una posición y una cantidad de bytes a leer, retorna 1) los bytes efectivamente leídos desde la posición pedida y la cantidad pedida en caso de ser posible, y 2) la cantidad de bytes que efectivamente se retornan leídos.

- **Escribir**: Dado un nombre de archivo, una cantidad de bytes determinada, y un buffer a partir del cual están los datos, se escriben los datos en el archivo dado. Si el archivo existe, los datos se agregan al final, si el archivo no existe, se crea y se le escriben los datos. En todos los casos se retorna la cantidad de bytes escritos.

Defina e implemente con gRPC un servidor.

[4] `https://github.com/github/gitignore/blob/master/Maven.gitignore`

3.2. Implementación

Proto A continuación se describe el archivo `proto`, el cual define el servicio `Ftp` con sus operaciones remotas.

Esta parte de la implementación será independiente del lenguaje de programación que se utilice, ya que gRPC generará una estructura lógica que estará escrita en el lenguaje que opte el programador (siempre y cuando sea soportado por el primero). Cabe destacar que los objetos remotos solo necesitarán conocer esta especificación para implementar y/o consumir un servicio.

En la Fig. 12 se puede ver el contenido del archivo `proto`, el cual está conformado por dos operaciones remotas:

```
1  syntax = "proto3";
2  package pdytr.example.grpc;
3
4  message ReadRequest {
5    string filename = 1;
6    int32 position = 2;
7    int32 offset = 3;
8  }
9
10 message ReadResponse {
11   bytes content = 1;
12   int32 bytes_read = 2;
13   bytes checksum = 3;
14 }
15
16 message WriteRequest {
17   string filename = 1;
18   int32 offset = 2;
19   bytes content = 3;
20   bool destroy_mode = 4;
21   bytes checksum = 5;
22 }
23
24 message WriteResponse {
25   int64 bytes_written = 1;
26   bool error = 2;
27 }
28
29 service Ftp {
30   rpc read(ReadRequest) returns (ReadResponse);
31   rpc write(WriteRequest) returns (WriteResponse);
32 }
```

Figura 12: Especificación protobuff del servicio FTP.

1. `read` (línea 30): Recibe como argumento el *message* `ReadRequest`, conformado por los siguientes campos:
   a) `filename` (línea 5): Indica el nombre del archivo que se desea leer.
   b) `position` (línea 6): Indica la posición de comienzo para leer.
   c) `offset` (línea 7): Indica la cantidad de bytes que se quieren leer.
   Retorna el *message* `ReadResponse`, que se compone por los campos:
   a) `content` (línea 11): El contenido del archivo en bytes.
   b) `bytes_read` (línea 12): La cantidad de bytes leídos.
   c) `checksum` (línea 13): El *checksum* generado a partir del contenido del archivo. Servirá para controlar que el contenido no se haya alterado en la transferencia.
2. `write` (línea 31): Recibe como argumento el *message* `WriteRequest`, el cual está conformado por los siguientes campos:
   a) `filename` (línea 17): Indica el nombre del archivo que se quiere escribir en el servidor.
   b) `offset` (línea 18): Indica una posición del archivo, que servirá como índice de partida para escribir el contenido.
   c) `content` (línea 19): Representa el contenido del archivo en bytes.
   d) `destroy_mode` (línea 20): Indica si la escritura debe realizarse en modo destructivo. Es de utilidad si se quiere sobreescribir un archivo existente en el servidor.
   e) `checksum` (línea 21): Indica el *checksum* generado a partir del contenido del archivo.
   Retorna el *message* `WriteResponse`, el cual está conformado por:
   a) `bytes_written` (línea 25): La cantidad totales de bytes escritos hasta el momento.
   b) `error` (línea 26): Indica si la comprobación del *checksum* fue errónea.

Cabe destacar que la línea 2 indica que el código generado por gRPC pertenecerá al paquete `pdytr.example.grpc` (En caso de ser compilado para *Java*, caso contrario se omite) [2].

Servidor El servidor se compone por las dos clases que se detallan a continuación:

1. `App`: Esta clase se encargará de escuchar las solicitudes de los clientes. La implementación es similar a la Fig. 7, solo se cambia la instanciación del servicio `Greeter` (línea 10) por el servicio `Ftp`.

2. `Ftp`: Esta clase implementará las operaciones remotas definidas en el archivo `proto`, a continuación se describen los métodos sobre-escritos:
   a) `read` (Ver Fig. 13): Se comienza abriendo un *stream* del archivo requerido por el cliente (líneas 4-5), y luego se lee el contenido desde la posición hasta el offset indicado (líneas 7-13).
   Posteriormente, se arma el *message* `ReadResponse` con el contenido, la cantidad de bytes leídos y el *checksum* generado (líneas 15-20), el cual es enviado a través del método `onNext()` (línea 22). Por último, se cierra el *stream* y la conexión gRPC (líneas 24-25).

- *Nota*: Se utiliza la clase `ByteString` (líneas 18 y 20) para transferir bytes, debido a que gRPC lo recomienda para asegurar la inmutabilidad de los datos [1].

```
1  @Override
2  public void read(ReadRequest request,
3              StreamObserver<ReadResponse> responseObserver) {
4      File file = new File(DB, request.getFilename());
5      FileInputStream stream = new FileInputStream(file);
6
7      int position = request.getPosition();
8      int offset = request.getOffset();
9
10     byte[] content = new byte[offset];
11
12     stream.skip(position);
13     int bytesRead = stream.read(content, 0, offset);
14
15     byte[] checksum = generateChecksum(content);
16
17     ReadResponse response = ReadResponse.newBuilder()
18             .setContent(ByteString.copyFrom(content))
19             .setBytesRead(bytesRead)
20             .setChecksum(ByteString.copyFrom(checksum)).build();
21
22     responseObserver.onNext(response);
23
24     stream.close();
25     responseObserver.onCompleted();
26 }
```

Figura 13: Implementación del servidor de la operación remota `read`.

b) `write` (Ver Fig. 14): Inicialmente se realiza la comprobación del *checksum* sobre el contenido recibido (línea 11):

- Si el resultado es exitoso, se abre el archivo en el modo indicado, es decir, destructivo o *append* (líneas 12-13). Luego, se escribe el contenido desde el final del archivo (línea 15) y por último, se envían la cantidad de bytes escritos a través del *message* `WriteResponse` (líneas 24-26).

- Si la comprobación es errónea, se elimina el archivo con el que se está trabajando y se lo comunica al cliente estableciendo `error=true` en el *message* `WriteResponse` (líneas 24-26).

```
1  @Override
2  public void write(WriteRequest request,
3              StreamObserver<WriteResponse> responseObserver) {
4      long bytesWritten = 0; boolean error = false;
5
6      File file = new File(DB, request.getFilename());
7
8      content = request.getContent();
9      checksum = request.getChecksum();
10
11     if (validChecksum(content, checksum)) {
12         boolean appendMode = !request.getDestroyMode();
13         FileOutputStream stream = new FileOutputStream(file, appendMode);
14
15         stream.write(content.toByteArray(), 0, request.getOffset());
16         stream.close();
17
18         bytesWritten = file.length();
19     } else {
20         error = true;
21         file.delete();
22     }
23
24     WriteResponse response = WriteResponse.newBuilder()
25         .setBytesWritten(bytesWritten)
26         .setError(error).build();
27
28     responseObserver.onNext(response);
29     responseObserver.onCompleted();
30 }
```

Figura 14: Implementación del servidor de la operación remota `write`.

Cliente El cliente es implementado a través de la clase `Client` con los siguientes métodos que utilizan las operaciones remotas descritas anteriormente:

1. `read` (Ver y ampliar Fig. 15): Inicialmente, se crea un archivo en donde se irá guardando los bytes recibidos por el servidor (líneas 2-3). Luego, comienza la iteración hasta leer el archivo por completo o hasta recibir un error de comprobación de *checksum* por el servidor. Durante cada iteración, se arma el *message* `ReadRequest` con los siguientes argumentos (líneas 8-11):
   a) El nombre del archivo que se quiere leer en el servidor (línea 9).
   b) La posición, que va a estar dada por la cantidad de bytes leídos hasta el momento (línea 10).
   c) La cantidad de bytes, que será una ventana de 100KB, la cual se mantendrá fija en todas las iteraciones (línea 11). En caso de que el tamaño disponible sea menor, el servidor leerá hasta dicho tamaño, por lo tanto, no habrá "valores basura" en el contenido retornado.

   Luego, se envía este *message* a través de la operación remota `read` (línea 13) y posteriormente se realizan las siguientes verificaciones (líneas 20 y 32):
   a) Si el *checksum* es válido y todavía hay bytes para leer, entonces se escribe el contenido recibido al final del archivo abierto por el cliente.
   b) Si la comprobación del *checksum* es errónea, entonces se termina la iteración y se elimina el archivo abierto.
   c) Si se llegó al fin del archivo, entonces termina la iteración.

```
1   private static void read(FtpBlockingStub stub, String filename) {
2       File file = new File(DB, filename);
3       FileOutputStream stream = new FileOutputStream(file);
4
5       boolean error = false; boolean eof = false; int totalBytesRead = 0;
6
7       while (!eof && !error) {
8           ReadRequest request = ReadRequest.newBuilder()
9                   .setFilename(filename)
10                  .setPosition(totalBytesRead)
11                  .setOffset(WINDOW).build();
12
13          ReadResponse response = stub.read(request);
14
15          int bytesRead = response.getBytesRead();
16          System.out.printf("Bytes read: %d\n", bytesRead);
17
18          eof = bytesRead == -1;
19
20          if (!eof) {
21              ByteString content = response.getContent();
22              ByteString checksum = response.getChecksum();
23
24              if (validChecksum(content, checksum)) {
25                  stream.write(content.toByteArray(), 0, bytesRead);
26                  totalBytesRead += bytesRead;
27              } else {
28                  System.err.println("Checksum is invalid.. abort");
29                  error = true;
30                  file.delete();
31              }
32          }
33      }
34
35      stream.close();
36  }
```

Figura 15: Implementación del cliente FTP que hace uso de la operación remota `read`.

2. `write` (Ver y ampliar Fig. 16): Para comenzar, se abre el archivo que se desea transferir al servidor (líneas 2-3), y se lo transferirá en porciones de 100KB (líneas 8-11). A medida que se lee el archivo se invoca a la operación remota `write` con un *message* `WriteRequest` conformado por los siguientes parámetros (líneas 15-20):

   a) El nombre del archivo que se desea escribir (línea 16).
   b) El contenido en bytes de la porción correspondiente del archivo (línea 17).
   c) El `offset`, que será el mínimo entre la ventana y la cantidad de bytes faltantes. Esto se debe al último fragmento del archivo, ya que si el tamaño no es múltiplo de la ventana entonces se estarían enviando "bytes basura" (línea 18).
   d) El modo de escritura, el cual en la primera iteración va a ser destructivo, mientras que en el resto de las iteraciones se irá agregando al final. Esto es de utilidad en el caso que exista un archivo con igual nombre en el servidor (línea 19).
   e) El *checksum* generado sobre el contenido a enviar (línea 20).

   Por último, la iteración se dará hasta que el total de bytes escritos en el archivo del servidor sea igual al tamaño del archivo en el cliente (línea 27) o hasta recibir un error por comprobación de *checksum* por parte del servidor (líneas 29-32).

```java
private static void write(FtpBlockingStub stub, String filename) {
    File file = new File(DB, filename);
    FileInputStream stream = new FileInputStream(file);

    boolean error = false; boolean eof = false; long bytesWritten = 0;

    while (!eof && !error) {
        byte[] content = new byte[WINDOW];

        int bytesRemaining = stream.available();
        stream.read(content);

        byte[] checksum = generateChecksum(content);

        WriteRequest request = WriteRequest.newBuilder()
                .setFilename(filename)
                .setContent(ByteString.copyFrom(content))
                .setOffset(min(WINDOW, bytesRemaining))
                .setDestroyMode(bytesWritten == 0)
                .setChecksum(ByteString.copyFrom(checksum)).build();

        WriteResponse response = stub.write(request);

        bytesWritten = response.getBytesWritten();
        System.out.printf("Bytes written: %d of %d\n", bytesWritten, file.length());

        eof = bytesWritten == file.length();

        if (response.getError()) {
            System.err.println("Checksum is invalid.. abort");
            error = true;
        }
    }

    stream.close();
}
```

Figura 16: Implementación del cliente FTP que hace uso de la operación remota `write`.

3.3. Ejecución

Automática Se desarrolló una ejecución a modo de ejemplo, en dónde se transfiere un archivo llamado `test.pdf` con un tamaño de 245KB desde el servidor hacia el cliente (utilizando la operación remota `read`), y luego desde el cliente al servidor (utilizando la operación remota `write`). Para ejecutar, se debe correr el siguiente comando por fuera del container:

```
docker exec --user root -it pdytr make -C /pdytr/java ftp
```

Manual En el caso de querer ejecutar el servidor FTP de forma manual, se deben ejecutar los siguientes comandos dentro del container:

```
# Ir al directorio fuente
cd /pdytr/java/ftp/

# Ejecutar el servidor
mvn -DskipTests package \
    exec:java \
    -Dexec.mainClass=pdytr.example.grpc.App

# Ejecutar el cliente (en otra sesión)
# Ir al directorio fuente
```

```
cd /pdytr/java/ftp/

# La base de datos del cliente se encuentra en db/client.
# La base de datos del servidor se encuentra en db/server.
# El archivo será leído/escrito en la "base de datos" correspondiente.
FILE="test.pdf"

# Para leer un archivo del servidor
mvn -DskipTests package \
    exec:java \
    -Dexec.mainClass=pdytr.example.grpc.Client \
    -Dexec.args="-read $FILE"

# Para escribir un archivo en el servidor
mvn -DskipTests package \
    exec:java \
    -Dexec.mainClass=pdytr.example.grpc.Client \
    -Dexec.args="-write $FILE"
```

4. Anexo: Python como lenguaje para gRPC

A continuación se presenta a Python como lenguaje para gRPC, utilizando como caso de estudio el sistema FTP descrito en la sección 3. El código fuente se encuentra en `src/python/ftp`.

Adicionalmente, por si la implementación resulta compleja, en el directorio `src/python/greeter` se encuentra la implementación del servidor `Greeter` descrito en la sección 2.2.

4.1. Gestor de dependencias

Para gestionar las dependencias en Python existen múltiples alternativas. Para este caso, se recomienda utilizar la herramienta *virtualenv* [8] debido a que permite encapsular el entorno de ejecución en un directorio, y podremos instalar las dependencias sin que afecten a todo el sistema. Los comandos más relevantes son los siguientes:

```
    # Creación del entorno virtual
    DIRNAME="venv"
    python3 -m venv $DIRNAME

    # Activación del entorno virtual
    source $DIRNAME/bin/activate

    # Desactivación del entorno virtual
    deactivate
```

4.2. Integración de gRPC

Primero deberemos instalar las dependencias necesarias para utilizar gRPC. Para ello, se utilizará *pip* [6], el gestor de paquetes estándar para Python. A continuación se muestra como instalarlas dentro del container:

```
DIRNAME="venv"

# Nos movemos al directorio fuente
cd /pdytr/python/
# Creamos el entorno virtual
python3 -m venv $DIRNAME
# Activamos el entorno virtual
source $DIRNAME/bin/activate
# Verificamos que pip esté en su última versión
pip install --upgrade pip
# Instalamos las dependencias
pip install -r requirements.txt
```

Cabe destacar que el contenido del archivo `requirements.txt` es el siguiente:

```
grpcio==1.39.0
grpcio-tools==1.39.0
```

Como se puede notar, son las dependencias necesarias para interactuar con gRPC. En caso de querer cambiar la versión, simplemente tendremos que modificar este archivo.

Luego, para generar el código de gRPC se debe correr el siguiente comando:

```
# Ir al directorio fuente
cd /pdytr/python/ftp/

# Generar el código
OUTPUT_DIR=.

python3 -m grpc_tools.protoc \
            -I. \
            --python_out=$OUTPUT_DIR \
            --grpc_python_out=$OUTPUT_DIR \
            ftp.proto
```

A continuación se explican los parámetros utilizados:

1. `-m grpc_tools.protoc`: Indica que se utiliza el módulo `grpc_tools.protoc`, el cual se encarga de generar el código especificado en el archivo `proto`.

2. `-I.`: Indica el directorio base para ejecutar los archivos por el módulo. En este caso, se toma el directorio actual.

3. `--python_out=$OUTPUT_DIR`: Indica el directorio en dónde generar el archivo `pdb2.py` (Se explica a continuación).

4. `--grpc_python_out=$OUTPUT_DIR`: Indica el directorio en dónde generar el archivo `pdb2_grpc.py` (Se explica a continuación).

5. `ftp.proto`: El archivo `proto` utilizado para la especificación del servicio FTP.

Este comando genera los archivos `ftp_pdb2.py` y `ftp_pdb2_grpc.py`. En el primero se encontrarán las clases que correspondan a los *messages* declarados en el archivo `proto`, mientras que en el segundo se encuentran funcionalidades para que utilicen el cliente y el servidor (stub, interfaz del servicio, etc.) [7].

Para simplificar la ejecución, todos estos pasos se encuentran en un *Makefile* ubicados en `src/python/Makefile`.

4.3. Implementación

Para implementar el servicio FTP se utiliza el algoritmo descrito en la sección 3.2 pero utilizando las características de Python como lenguaje. A continuación se explica cada componente.

Proto El archivo proto que se utiliza es igual al utilizado en la implementación de Java (Ver Fig. 12). Como se dijo anteriormente, este componente es independiente del lenguaje.

Servidor Tal como se puede apreciar en la Fig. 17, la función `serve` (línea 31) se encargará de escuchar las solicitudes de los clientes en el puerto indicado por parámetro, y la clase `Ftp` (línea 1) es la implementación del servicio FTP declarado en el archivo `proto`. La misma, implementa las operaciones remotas `read` y `write` del mismo modo que la descrita en Java (Ver sección 3.2).

Cliente De la implementación del cliente se puede destacar el uso de las operaciones remotas `read` (línea 1) y `write` (línea 8) (Ver Fig. 18). Las mismas se invocan a través de la estructura `with` [5], la cual nos asegura de terminar correctamente la conexión con el canal. Luego, el algoritmo es el mismo que el descrito en la sección 3.2, pero utilizando la sintaxis de Python, en caso de querer ver la implementación completa mirar el archivo `src/python/ftp/client.py`.

4.4. Ejecución

Automática Para una ejecución similar a la descrita en la sección 3.3 se debe correr el siguiente comando fuera del container:

```
docker exec --user root -it pdytr make -C /pdytr/python ftp
```

```python
class Ftp(FtpServicer):
    def read(self, request, context):
        with open(DB / request.filename, "rb") as file:
            file.seek(request.position)
            content = file.read(request.offset)

        return ReadResponse(
            content=content,
            bytes_read=-1 if content == EOF else len(content),
            checksum=self._generate_checksum(content),
        )

    def write(self, request, context):
        file_path = DB / request.filename
        bytes_written = 0

        if error := not self._valid_checksum(request):
            file_path.unlink(missing_ok=True)
        else:
            mode = "wb" if request.destroy_mode else "ab"

            with open(file_path, mode) as file:
                file.seek(request.offset)
                file.write(request.content)

            bytes_written = file_path.stat().st_size

        return WriteResponse(bytes_written=bytes_written, error=error)


def serve(port):
    logging.basicConfig()

    server = grpc.server(ThreadPoolExecutor(max_workers=MAX_CONNECTIONS))
    add_FtpServicer_to_server(Ftp(), server)
    server.add_insecure_port(f"[::]:{port}")
    server.start()
    server.wait_for_termination()
```

Figura 17: Implementación del servidor FTP en Python.

```python
def read_remote(port, payload):
    with grpc.insecure_channel(f"localhost:{port}") as channel:
        request = ReadRequest(**payload)
        response = FtpStub(channel).read(request)

    return response

def write_remote(port, payload):
    with grpc.insecure_channel(f"localhost:{port}") as channel:
        request = WriteRequest(**payload)
        response = FtpStub(channel).write(request)

    return response
```

Figura 18: Operaciones remotas del cliente FTP en Python.

Manual Para ejecutar manualmente seguir los siguientes comandos dentro del container:

```
# Ir al directorio fuente
cd /pdytr/python/ftp/

DIRNAME="venv"
PORT="50051"

# Activamos el entorno virtual
source /pdytr/python/$DIRNAME/bin/activate

# Ejecutamos el servidor
python3 server.py --port $PORT

# Ejecutamos el cliente (en otra sesión)
# Ir al directorio fuente
cd /pdytr/python/ftp/

DIRNAME="venv"
PORT="50051"
FILE="test.pdf"

# Activamos el entorno virtual
source /pdytr/python/$DIRNAME/bin/activate

# Para lectura
python3 client.py --read $FILE --port $PORT

# Para escritura
python3 client.py --write $FILE --port $PORT
```

Referencias

1. ByteString, https://developers.google.com/protocol-buffers/docs/reference/java/com/google/protobuf/ByteString
2. Language Guide (proto3) | Protocol Buffers | Google Developers, https://developers.google.com/protocol-buffers/docs/proto3
3. Maven – Maven Documentation, https://maven.apache.org/guides/
4. Maven – POM Reference, https://maven.apache.org/pom.html
5. PEP 343 – The "with"Statement | Python.org, https://www.python.org/dev/peps/pep-0343/
6. pip · PyPI, https://pypi.org/project/pip/
7. Python | gRPC, https://grpc.io/docs/languages/python/
8. Virtualenv · PyPI, https://virtualenv.pypa.io/en/latest/