

Practica 3

Inti María Tidball 17612/3

Juan Pablo Sanchez Magariños 13238/3

1. Utilizando como base el programa ejemplo1 de gRPC: Mostrar experimentos donde se produzcan errores de conectividad del lado del cliente y del lado del servidor.

a. Si es necesario realice cambios mínimos para, por ejemplo, incluir `exit()`, de forma tal que no se reciban comunicaciones o no haya receptor para las comunicaciones.

i. Si el cliente se corre sin una respuesta del servidor (no se levanta o se hace un `exit` temprano) el cliente arroja:

RuntimeException: io.grpc.StatusRuntimeException: UNAVAILABLE

Corrimos el cliente sin antes iniciar el servidor:

```
podman exec grpc mvn -DskipTests package exec:java  
-Dexec.mainClass=pydytr.example.grpc.Client
```

```
Downloaded from central: https://repo.maven.apache.org/maven2/org/codehaus/plexus/plexus-utils/3.4.2/plexus-utils-3.4.2.jar  
Downloaded from central: https://repo.maven.apache.org/maven2/org/apache/commons/commons-exec/1.3/commons-exec-1.3.jar (54  
[WARNING]  
io.grpc.StatusRuntimeException: UNAVAILABLE  
    at io.grpc.stub.ClientCalls.toStatusRuntimeException (ClientCalls.java:210)  
    at io.grpc.stub.ClientCalls.getUnchecked (ClientCalls.java:191)  
    at io.grpc.stub.ClientCalls.blockingUnaryCall (ClientCalls.java:124)  
    at pydytr.example.grpc.GreetingServiceGrpc$GreetingServiceBlockingStub.greeting (GreetingServiceGrpc.java:163)  
    at pydytr.one.a.i.Client.main (Client.java:28)  
    at org.codehaus.mojo.exec.ExecJavaMojo$1.run (ExecJavaMojo.java:279)  
    at java.lang.Thread.run (Thread.java:748)  
Caused by: io.netty.channel.AbstractChannel$AnnotatedConnectException: Connection refused: localhost/0:0:0:0:0:0:1:8080  
    at sun.nio.ch.SocketChannelImpl.checkConnect (Native Method)  
    at sun.nio.ch.SocketChannelImpl.finishConnect (SocketChannelImpl.java:716)  
    at io.netty.channel.socket.nio.NioSocketChannel.doFinishConnect (NioSocketChannel.java:323)  
    at io.netty.channel.nio.AbstractNioChannel$AbstractNioUnsafe.finishConnect (AbstractNioChannel.java:340)
```

ii. Agregando un `System.exit()` en el archivo "`GreetingServiceImpl.java`" se obtiene el mismo error **UNAVAILABLE**. En este caso se puede correr utilizando los scripts proporcionados. Estos se encargan de crear los contenedores y correr los comandos de maven en los mismos, en distintas terminales correr:

```
$ ./start.sh && ./runapp.sh one.a
```

```
$ ./runclient.sh one.a
```

```
Downloaded from central: https://repo.maven.apache.org/maven2/org/apache/commons/commons-exec/1.3/commons-exec-1.3.jar  
Downloaded from central: https://repo.maven.apache.org/maven2/org/codehaus/plexus/plexus-utils/3.4.2/plexus-utils-3.4.2.jar  
Server started  
[INFO] BUILD FAILURE  
[INFO] Total time: 19.611 s  
[INFO] Finished at: 2023-10-26T05:47:39Z  
[ERROR] Failed to execute goal org.codehaus.mojo:exec-maven-plugin:3.1.0:java (default-cli) on project grpc: The Java class. UNAVAILABLE: Network closed for unknown reason -> [Help 1]  
[ERROR] To see the full stack trace of the errors, re-run Maven with the -e switch.  
[ERROR] Re-run Maven using the -X switch to enable full debug logging.  
[ERROR] For more information about the errors and possible solutions, please read the following articles:  
[ERROR] [Help 1] http://cwiki.apache.org/confluence/display/MAVEN/MojoExecutionException
```

- b. Configure un **DEADLINE** y cambie el código (agregando la función *sleep()*) para que arroje la excepción correspondiente.

Luego de hacer estas modificaciones, se obtiene un error de tipo **DEADLINE_EXCEEDED**

```
./start.sh && ./runapp.sh one.b  
./runclient.sh one.b
```

```
Downloaded from central: https://repo.maven.apache.org/maven2/org/codehaus/plexus/plexus-utils/3.4.2/plexus-utils-3.4.2.jar
Downloaded from central: https://repo.maven.apache.org/maven2/org/apache/commons/commons-exec/1.3/commons-exec-1.3.jar
Server started

[INFO] --- maven-compiler-plugin:3.8.0:testCompile (default-testCompile) @ grpc-hello-server ---
[INFO] Nothing to compile - all classes are up to date
[INFO] --- maven-surefire-plugin:2.12.4:test (default-test) @ grpc-hello-server ---
[INFO] Tests are skipped.
[INFO] --- maven-jar-plugin:2.4:jar (default-jar) @ grpc-hello-server ---
[INFO] Building jar: /grpc-hello-server/target/grpc-hello-server-1.0-SNAPSHOT.jar
[INFO] --- exec-maven-plugin:3.1.0:java (default-cli) @ grpc-hello-server ---
[WARNING]
io.grpc.StatusRuntimeException: DEADLINE_EXCEEDED: deadline exceeded after 2975958373ns
    at io.grpc.stub.ClientCalls.toStatusRuntimeException (ClientCalls.java:210)
    at io.grpc.stub.ClientCalls.getUnchecked (ClientCalls.java:191)
    at io.grpc.stub.ClientCalls.blockingUnaryCall (ClientCalls.java:124)
    at pdytr.example.grpc.GreetingServiceGrpc$GreetingServiceBlockingStub.greeting (GreetingServiceGrpc.java:163)
    at pdytr.one.b.Client.main (Client.java:29)
    at org.codehaus.mojo.exec.ExecJavaMojo$1.run (ExecJavaMojo.java:279)
    at java.lang.Thread.run (Thread.java:748)
```

- c. Reducir el *deadline* de las llamadas gRPC a un 10% menos del promedio encontrado anteriormente. Mostrar y explicar el resultado para 10 llamadas.

Establecimos un *deadline* de 5000 milisegundos en el cliente y un *sleep* de 4500 ms. Encontramos que la primera llamada se produce el error antes obtenido, pero en las 9 siguientes termina correctamente. Realizamos varias veces el experimento obteniendo resultados similares: fallar las dos primeras veces y luego 8 correctas, o fallar las 3 primeras y 7 correctas. Llegamos a la conclusión de que en la primera comunicación el servidor tarda un tiempo adicional en responder.

2. Describir y analizar los tipos de API que tiene gRPC.

gRPC admite cuatro tipos de API:

Unary RPCs: El cliente envía una petición simple, es decir una sola, y recibe una sola respuesta.

Server streaming RPCs: Se envía una petición simple, pero se recibe una secuencia de respuestas, incluido un status y metadatos del servidor.

Client streaming RPCs: Una secuencia de peticiones es enviada, y se recibe una respuesta simple.

Bidirectional streaming RPCs: El cliente inicia una llamada indicando un método, metadatos y un *deadline*. Luego el servidor elige si espera una secuencia de mensajes o reenviar los metadatos iniciales.

Desarrolle una conclusión acerca de cuál es la mejor opción para los siguientes escenarios:

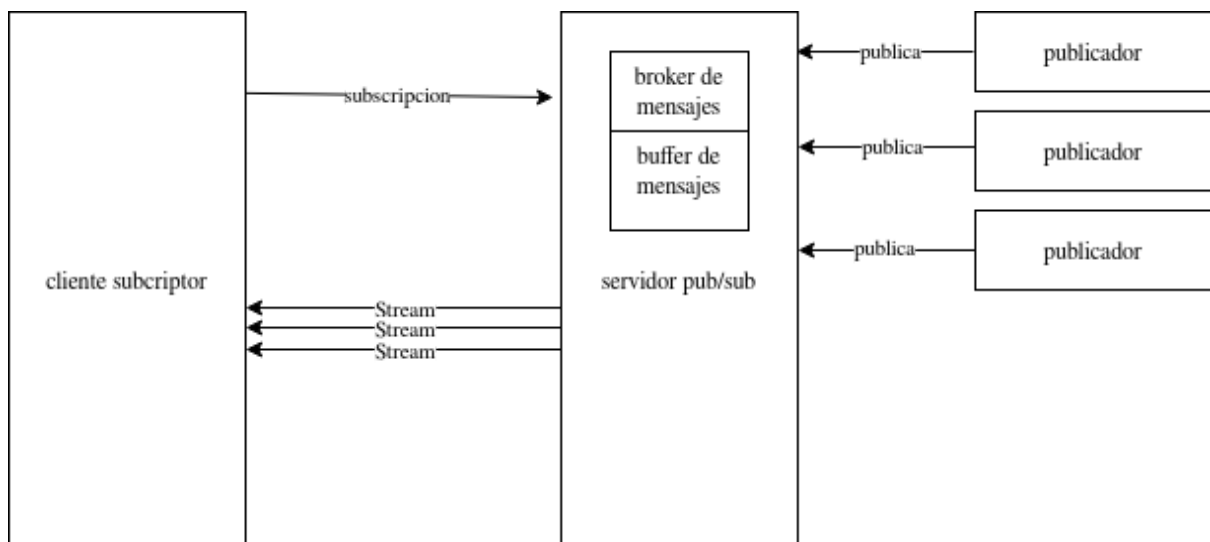
Un sistema de pub/sub

En un sistema de pub/sub, los suscriptores se suscriben a ciertos temas y los publicadores publican mensajes en esos temas. Cuando un mensaje es publicado en un tema, se envía a todos los suscriptores de ese tema.

La eficiencia en la entrega de mensajes es clave. Los Server Streaming RPCs son ideales para este escenario porque una vez que un suscriptor indica su interés en un tema mediante una solicitud inicial, el servidor puede enviar mensajes de forma continua sin requerir solicitudes adicionales. Esto reduce la latencia y el overhead de la red al eliminar la necesidad de múltiples solicitudes y respuestas para obtener los mensajes. Además, esta aproximación se alinea con el modelo de comunicación "push", que es intrínseco a los sistemas pub/sub, permitiendo que los mensajes lleguen a los suscriptores tan pronto como están disponibles sin que los suscriptores necesiten sondear activamente el servidor.

Por eso, podemos usar **Server streaming RPCs**

- Cuando un suscriptor se suscribe a un tema, realiza una solicitud al servidor indicando su interés. A partir de ese momento, el servidor tiene la capacidad de enviar continuamente mensajes a ese suscriptor sin que el suscriptor tenga que hacer solicitudes adicionales.
- Una vez establecida la conexión, el servidor puede "pushear" mensajes hacia el suscriptor cada vez que haya un nuevo mensaje en el tema al que se ha suscrito. Los suscriptores no "tiran" de los mensajes; en su lugar, los mensajes son "empujados" hacia ellos por el publicador (o en este caso, el servidor).



Un sistema de archivos FTP

Un sistema FTP permite a los usuarios cargar y descargar archivos. Las operaciones típicas incluyen listar archivos, cargar un archivo, descargar un archivo, entre otros. Las operaciones de listado y descarga de archivos pueden implementarse eficientemente con Synchronous Unary RPCs, debido a su simplicidad y naturaleza de petición-respuesta. Para la descarga de archivos grandes, la técnica de chunking es esencial, permitiendo que

los archivos se transfieran en segmentos manejables. Esto no solo facilita el manejo de memoria y la transferencia de datos, sino que también permite una recuperación más fácil en caso de interrupciones, ya que se puede reanudar la descarga desde el último segmento recibido correctamente. Para la carga de archivos, los Client Streaming RPCs son preferibles, ya que permiten el envío de datos de manera secuencial y continua, lo cual es beneficioso para el manejo de archivos grandes y proporciona la posibilidad de monitorear el progreso de la carga en tiempo real.

Usamos **Synchronous Unary RPCs & Client streaming RPCs**

- Para listar y descargar archivos podríamos usar Unary RPCs, y cuando un usuario quiere listar los archivos en un directorio o descargar un archivo específico, realiza una petición y espera una respuesta. Esta es una operación simple de petición-respuesta, que si es necesario se puede iterar. Unary RPCs es ideal para estas operaciones porque es directo y eficiente para transacciones individuales.
- La descarga de archivos más grandes usa chunking. Se puede tener un estado que almacena el lugar donde está, el offset, y el byte final.
- Para subir archivos podríamos usar Client streaming RPCs ya que la subida de archivos, especialmente archivos grandes, puede ser más compleja. En lugar de enviar todo el archivo en una sola petición que es propenso a errores podemos enviarlo en fragmento con streaming secuencialmente, lo cual también ofrece la posibilidad de monitorear el progreso de la subida.
- La reanudación de transferencias son necesarias para mitigar interrupciones de conectividad.

API:

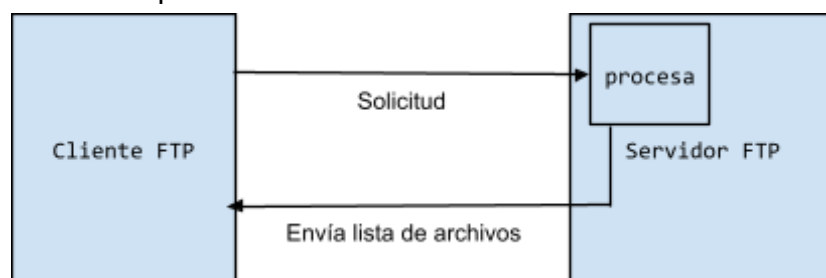
1. Listar Archivos:

Cliente FTP:

- Inicia la aplicación y selecciona la opción "Listar Archivos".
- Envía una solicitud RPC unaria al Servidor FTP para obtener la lista de archivos disponibles.

Servidor FTP:

- Recibe la solicitud RPC unaria y consulta la base de datos o el sistema de archivos para obtener la lista.
- Genera una respuesta que contiene los nombres de los archivos, tamaños y fechas de creación.
- Envía la respuesta al Cliente FTP.



2. Descargar Archivo:

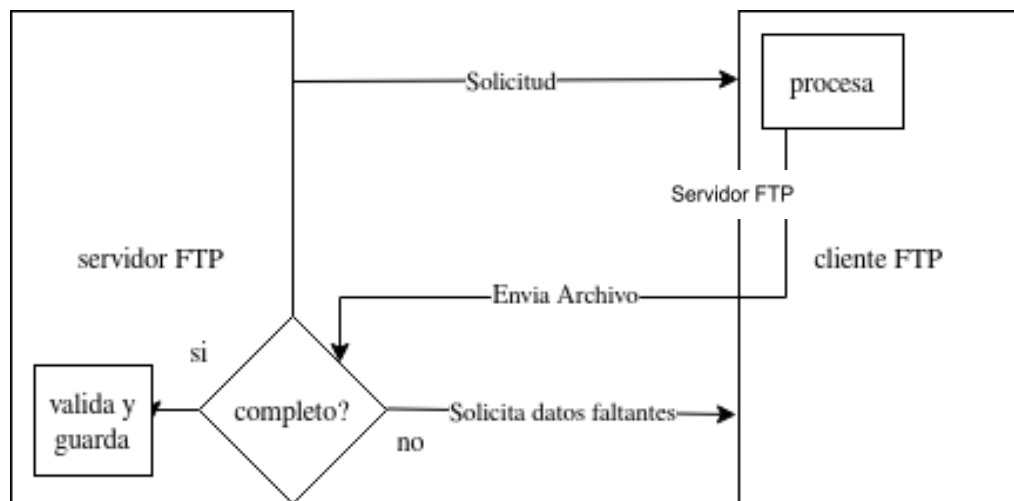
Cliente FTP:

- El usuario selecciona un archivo de la lista proporcionada y elige "Descargar".

- Envía una solicitud RPC unaria al Servidor FTP para descargar el archivo especificado.
- Recibe los datos del archivo. Si el archivo es grande, el cliente lo maneja en trozos, ensamblando los trozos conforme llegan.
- Verifica la integridad del archivo descargado, posiblemente mediante un checksum o a lo sumo un diff.

Servidor FTP:

- Recibe la solicitud de descarga y busca el archivo en el almacenamiento.
- Divide el archivo en trozos si es grande y los envía secuencialmente.
- Envía un mensaje de finalización después del último trozo para indicar que la transmisión ha terminado



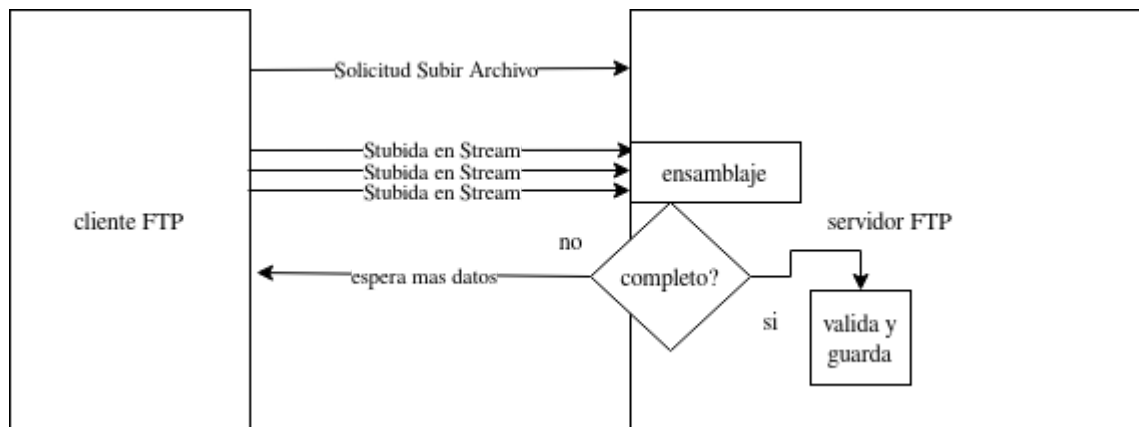
3. Subir Archivo:

Cliente FTP:

- El usuario elige "Subir Archivo" y selecciona un archivo desde su dispositivo.
- Inicia un RPC de streaming del cliente hacia el Servidor FTP y comienza a enviar el archivo en trozos.
- Envía un checksum del archivo completo para la verificación posterior por parte del servidor.
- Recibe una confirmación del servidor una vez que la carga está completa y verificada.

Servidor FTP:

- Prepara un nuevo archivo en el almacenamiento para recibir los datos.
- Recibe los trozos del archivo y los ensambla conforme llegan.
- Verifica la integridad del archivo ensamblado usando el checksum proporcionado.
- Envía una respuesta al Cliente FTP confirmando la recepción y la validez del archivo subido.

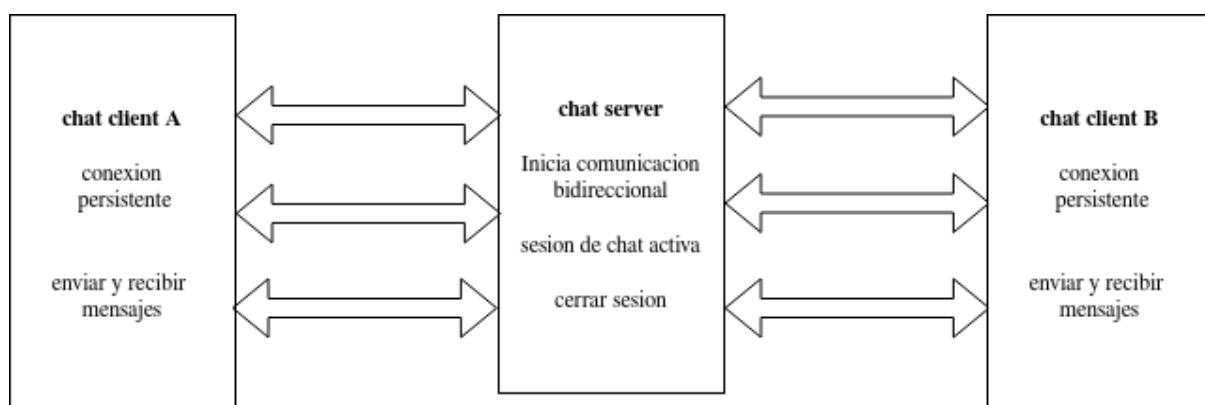


Un sistema de chat

En un chat, los usuarios se comunican en tiempo real, y no es eficiente enviar un mensaje para esperar su respuesta. Por naturaleza del sistema, se espera un flujo bidireccional de datos, los usuarios envían y reciben datos continuamente. Los Bidirectional Streaming RPCs son la mejor opción aquí, ya que permiten un flujo de mensajes bidireccional y continuo sobre una única conexión persistente. Este método es más eficiente que establecer nuevas conexiones para cada interacción, lo cual sería costoso en términos de recursos y tiempo. Además, una conexión bidireccional permite que los mensajes se envíen y reciban de manera asíncrona para la interactividad de un chat en vivo.

Uso de Bidirectional Streaming RPCs:

- Se establece una conexión que es persistente y permite comunicación en tiempo real.
- Se permite una comunicación de dos vías con una sola conexión sin bloqueos
- Se evita la sobrecarga de establecer nuevas conexiones continuamente, que es la parte más costosa de la comunicación



3. Analizar la transparencia de gRPC en cuanto al manejo de parámetros de los procedimientos remotos. Considerar lo que sucede en el caso de los valores de retorno. Puede aprovechar el ejemplo provisto.

En relación a la transparencia del protocolo gRPC, el tipo de parámetros que se esperan en los requests, están claramente definidos en el archivo .proto que se utiliza para

definir el servicio. El archivo .proto es esencialmente la interfaz entre el cliente y el servidor en gRPC.

El archivo .proto se compila usando el compilador de Protocol Buffers y puede generar código para diferentes lenguajes de programación que podrán interactuar de manera transparente ya que comparten el mismo protocolo. Las clases generadas proporcionarán métodos para establecer y obtener los valores de los campos de los mensajes, así como para invocar las funciones RPC en el cliente y para implementarlas en el servidor.

Comparación con swagger para API REST

El archivo .proto en gRPC es similar a una especificación Swagger (OpenAPI) para una API REST. Ambos proporcionan una definición estructurada y legible por máquinas de la interfaz de un servicio web. Aquí hay algunas similitudes y diferencias clave:

Similitudes:

Tanto Swagger (OpenAPI) como .proto describen la interfaz de un servicio web. Definen qué operaciones están disponibles, qué parámetros esperan, y qué devuelven.

A partir de la especificación Swagger o del archivo .proto, puedes generar automáticamente código cliente en varios lenguajes de programación.

Ambos pueden ser utilizados para generar documentación interactiva para el servicio. Por ejemplo, Swagger UI permite explorar y probar una API REST directamente desde la documentación basada en la especificación Swagger. Del mismo modo, hay herramientas que pueden generar documentación para servicios gRPC a partir de archivos .proto.

Diferencias:

Mientras que Swagger (OpenAPI) está específicamente diseñado para describir APIs REST que utilizan HTTP/1.1 como protocolo, gRPC utiliza Protocol Buffers como lenguaje de definición de interfaz y HTTP/2 como protocolo de transporte.

Swagger (OpenAPI) generalmente describe APIs que intercambian datos en formato JSON, mientras que gRPC utiliza Protocol Buffers, que es un formato binario.

Las APIs REST descritas por Swagger operan generalmente en términos de verbos HTTP (GET, POST, PUT, DELETE, etc.) y recursos. gRPC, por otro lado, define procedimientos remotos (RPCs) que pueden ser más abstractos y no se atan a los conceptos RESTful tradicionales.

4. 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.

a. Defina e implemente con gRPC un servidor. Documente todas las decisiones tomadas.

Se definió la interfaz **FtpService.proto** que se implementa luego en el servicio **FtpServiceImpl**. Se definen los métodos **read()** y **write()**, implementando una api Unary, ya que se recibirá una petición y se enviará una respuesta.

Los mensajes definidos son:

ReadRequest: que el cliente utilizará para solicitar la lectura de un archivo. Contiene un string que representa el nombre del archivo, dos enteros, uno que indica la posición para iniciar la lectura y el otro la cantidad de bytes a leer

WriteRequest: con el que se solicita la escritura y también tiene un string para el nombre del archivo, la cantidad de bytes que se van a escribir en un entero y un campo de datos, de tipo ByteString.

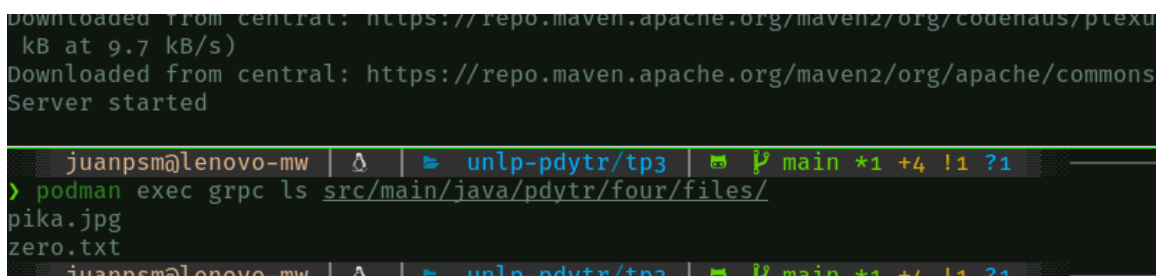
ReadResponse: retorna los datos leídos en un ByteString y un entero que indica la cantidad de bytes leídos.

WriteResponse: retorna la cantidad de bytes escritos.

Para correr el ejemplo se inicia el servidor con los scripts, en este caso el argumento **four** hace referencia a este ejercicio:

```
./start.sh && ./runapp.sh four
```

Luego, para verificar vemos los archivos que existen en directorio **files** del contenedor, ya que luego de correr el cliente se debe crear un nuevo archivo



```
Downloaded from central: https://repo.maven.apache.org/maven2/org/codenaus/ptextu
kB at 9.7 kB/s)
Downloaded from central: https://repo.maven.apache.org/maven2/org/apache/commons
Server started

juanpsma@lenovo-mw | ^ | unlp-pdytr/tp3 | P main *1 +4 !1 ?1
> podman exec grpc ls src/main/java/pdytr/four/files/
pika.jpg
zero.txt
juanpsma@lenovo-mw | ^ | unlp-pdytr/tp3 | P main *1 +4 !1 ?1
```

En otra terminal corremos el cliente, diciéndole que archivo leer. Nuestro cliente pide al servidor leer el archivo indicado. Si el mismo no existe, devuelve una respuesta vacía. Por ejemplo si se corre:

```
./runclient.sh four non-existent.txt
```



```
[INFO] Building jar: /grpc-hello-server/target/grpc-hello-server-1.0-SNAPSHOT.jar
[INFO]
[INFO] — exec-maven-plugin:3.1.0:java (default-cli) @ grpc-hello-server —
io.grpc.StatusRuntimeException: INTERNAL: Completed without a response
    at io.grpc.stub.ClientCalls.toStatusRuntimeException(ClientCalls.java:210)
    at io.grpc.stub.ClientCalls.getUnchecked(ClientCalls.java:191)
    at io.grpc.stub.ClientCalls.blockingUnaryCall(ClientCalls.java:124)
    at pdytr.four.FtpServiceGrpc$FtpServiceBlockingStub.read(FtpServiceGrpc.java:174)
    at pdytr.four.Client.main(Client.java:55)
    at org.codehaus.mojo.exec.ExecJavaMojo$1.run(ExecJavaMojo.java:279)
    at java.lang.Thread.run(Thread.java:748)
[INFO]
[INFO] BUILD SUCCESS
[INFO]
[INFO] Total time: 7.506 s
[INFO] Finished at: 2023-11-16T05:18:27Z
```

Si lo corremos con un archivo que existe:

```
./runclient.sh four zero.txt
```

Entonces el servidor responde con el contenido del archivo. Para simplificar la demostración, en el momento que el cliente recibe una respuesta, hace una petición para escribir un archivo llamado “**output**” al servidor con el contenido de lo que acaba de recibir como respuesta.

De esta manera probamos todos los métodos implementados en una sola ejecución.

```
Downloaded from central: https://repo.maven.apache.org/maven2/org/codehaus/plexus/plexus-component-anno
kB at 9.7 kB/s)
Downloaded from central: https://repo.maven.apache.org/maven2/org/apache/commons/commons-exec/1.3/commo
Server started

[INFO]
[INFO] — exec-maven-plugin:3.1.0:java (default-cli) @ grpc-hello-server —
[INFO]
[INFO] BUILD SUCCESS
[INFO]
[INFO] Total time: 7.567 s
[INFO] Finished at: 2023-11-16T05:03:11Z
[INFO]

juanpsma@lenovo-mw | Δ | unlp-pdytr/tp3 | P main *1 +4 !1 ?1
> podman exec grpc ls src/main/java/pdytr/four/files/
output
pika.jpg
zero.txt
juanpsma@lenovo-mw | Δ | unlp-pdytr/tp3 | P main *1 +4 !1 ?1
> podman exec grpc cat src/main/java/pdytr/four/files/output
oooooooooooooooooooooooooooooooooooooooooooooooooooo
juanpsma@lenovo-mw | Δ | unlp-pdytr/tp3 | P main *1 +4 !1 ?1
> podman exec grpc diff src/main/java/pdytr/four/files/zero.txt src/main/java/pdytr/four/files/output
juanpsma@lenovo-mw | Δ | unlp-pdytr/tp3 | P main *1 +4 !1 ?1
```

Podemos ver que se ha creado un archivo llamado “**output**” con el mismo contenido que “**zero.txt**” lo que se puede corroborar con:

```
podman exec grpc cat src/main/java/pdytr/four/files/output
podman exec grpc diff src/main/java/pdytr/four/files/zero.txt \
src/main/java/pdytr/four/files/output
```

Se puede volver a correr el cliente con el mismo archivo y de esa forma veremos como se concatena en el mismo output, dado que por requerimiento si el archivo a escribir existe, el servidor debe agregar al final el contenido.

```
[INFO] Building jar: /grpc-hello-server/target/grpc-hello-server-1.0-SNAPSHOT.jar
[INFO]
[INFO] — exec-maven-plugin:3.1.0:java (default-cli) @ grpc-hello-server —
[INFO]
[INFO] BUILD SUCCESS
[INFO]
[INFO] Total time: 7.467 s
[INFO] Finished at: 2023-11-16T05:05:05Z
[INFO]
juanpsm@lenovo-mw | Δ | ➤ unlp-pdytr/tp3 | 🐛 🔍 main *1 +4 !1 ?1
➤ podman exec grpc diff src/main/java/pdytr/four/files/zero.txt src/main/java/pdytr/four/files/output
— src/main/java/pdytr/four/files/zero.txt
++ src/main/java/pdytr/four/files/output
@@ -1 +1 @@
-ooooooooooooooooooooooooooooooooooooooooooooo
\ No newline at end of file
+ooooooooooooooooooooooooooooooooooooooooooooo
\ No newline at end of file
juanpsm@lenovo-mw | Δ | ➤ unlp-pdytr/tp3 | 🐛 🔍 main *1 +4 !1 ?1
```

Esto no solo funciona con archivos de texto plano, también podemos probar un binario, en este caso tenemos un JPG:

```
./runclient.sh four pika.jpg
```

Como ya existe nuestro output vemos que se ha concatenado el contenido.

[illegible]

Para verificar que el archivo se reproduce completamente hay que reiniciar servidor (para que no adicione en output) entonces se cancela en la terminal donde corría el servidor y otra vez se puede recrear todo con:

```
./start.sh && ./runapp.sh four
```

Y en otra terminal el cliente

```
./runclient.sh four pika.jpg
```

Con `diff` se puede corroborar que el archivo se leyó y se volvió a escribir correctamente:

[illegible]

- b. Investigue si es posible que varias invocaciones remotas estén ejecutándose concurrentemente y si esto es apropiado o no para el servidor de archivos del ejercicio anterior. En caso de que no sea apropiado, analice si es posible proveer una solución (enunciar/describir una solución, no es necesario implementarla).

Nota: diseñe un experimento con el que se pueda demostrar fehacientemente que dos o más invocaciones remotas se ejecutan concurrentemente o no.

Como gRPC utiliza **HTTP/2**, se permite la multiplexación de múltiples llamadas RPC sobre una única conexión. Esto significa que se pueden tener múltiples flujos de datos (streams) en una sola conexión, y que las llamadas RPC se ejecuten concurrentemente sin tener que establecer múltiples conexiones. Por lo tanto, en términos de transporte y comunicación, gRPC es capaz de manejar múltiples solicitudes y respuestas concurrentemente gracias a HTTP/2.

Sin embargo esto puede ser problemático en un servidor FTP si dos o más usuarios quieren escribir el mismo archivo. Las peticiones de lectura pueden ser concurrentes y asincrónicas, pero las de escritura deberían ser restringidas a un solo usuario por recurso y sincronizadas.

Se puede usar un `async read` en el proto:

```
rpc AsyncRead(FileRequest) returns (stream FileResponse);
```

Para la escritura se podrían usar locks.

A modo de experimento agregamos otro archivo a nuestro servidor FTP, one.txt, que tiene todos unos. Veremos que sucede cuando se corren dos clientes al mismo tiempo, uno que escriba en output el archivo zero.txt antes visto y el otro escribe en el mismo archivo de salida el nuevo one.txt.

Para ejecutar ambos clientes al mismo tiempo nos valemos de la herramienta [parallel](#), y creamos el script `runparallel.sh` que nos facilita la ejecución.

Entonces corremos otra vez el servidor en una terminal

```
./start.sh && ./runapp.sh four
```

Y en otra terminal los clientes

```
./runparallel.sh
```

```
Downloaded from central: https://repo.maven.apache.org/maven2/org/apache/commons/commons-e
Server started

[INFO] --- maven-compiler-plugin:3.8.0:testCompile (default-testCompile) @ grpc-hello-serv
[INFO] Nothing to compile - all classes are up to date
[INFO]
[INFO] --- maven-surefire-plugin:2.12.4:test (default-test) @ grpc-hello-server ---
[INFO] Tests are skipped.
[INFO]
[INFO] --- maven-jar-plugin:2.4:jar (default-jar) @ grpc-hello-server ---
[INFO]
[INFO] --- exec-maven-plugin:3.1.0:java (default-cli) @ grpc-hello-server ---
[INFO]
[INFO] BUILD SUCCESS
[INFO]
[INFO] Total time: 12.137 s
[INFO] Finished at: 2023-11-16T07:30:27Z
[INFO]
juanpsm@lenovo-mw | Δ | ▶ unlp-pdytr/tp3 | 🐛 📄 main *1 +4 !2 ?3
> podman exec grpc cat src/main/java/pdytr/four/files/output
110000011001100110000110011001111001100110011001111001100001111001100110011001100%
juanpsm@lenovo-mw | Δ | ▶ unlp-pdytr/tp3 | 🐛 📄 main *1 +4 !2 ?3
```

Vemos cómo se mezclan ambos archivos leídos en la salida.

5. Tiempos de respuesta de una invocación

- Diseñe un experimento que muestre el tiempo de respuesta mínimo de una invocación con gRPC. Muestre promedio y desviación estándar de tiempo respuesta.
- Utilizando los datos obtenidos en la Práctica 1 (Socket) realice un análisis de los tiempos y sus diferencias. Desarrollar una conclusión sobre los beneficios y complicaciones tiene una herramienta sobre la otra.

Se desarrolla un **TimeService.proto** con arquitectura cliente-servidor donde el request guarda un timestamp y una cantidad de bytes enviados.

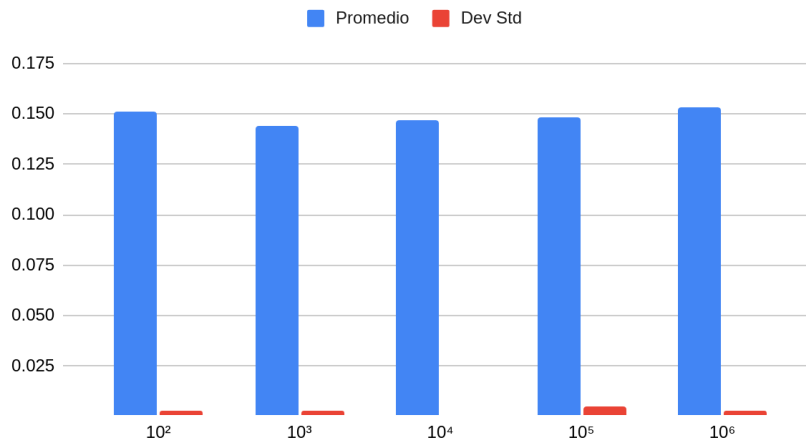
En la implementación, el método **getTime** imprime la cantidad de bytes enviados para poder verificar la correctitud.

En el cliente se recibe el parámetro y se crea un byte array del tamaño del parámetro enviado.

Se mide el tiempo previo al request y también luego del response. Esto se hace en el cliente replicando la metodología de la práctica 1, Se divide por 2 para tener un promedio de tiempo para el envío o la respuesta.

Se ve que los tiempos mínimos son en el envío de 1000 a 10000 bytes. Menos que eso genera un pequeño overhead. La desviación estándar es mínima.

Promedio and Dev Std



Usamos un script donde se puede pasar la cantidad de bytes, por ejemplo:

```
./start.sh && ./runapp.sh five
```

```
./runclient.sh five 4096
```

```
Downloaded from central: https://repo.maven.apache.org/maven2/org/codehaus/plexus/plexus-utils/3.4.2/plexus-utils-3.4.2.jar (267 kB at 2.0 MB/s)
Server started, listening on 50051
Received data size: 4096 bytes
```

```
[INFO] --- exec-maven-plugin:3.1.0:java (default-cli) @ grpc-hello-server ---
trip time: 0.1510 seconds
Server timestamp: 1699578607032
```

El experimento se corre 5 veces para cada valor de bytes, de 10^2 a 10^6 . En 10^7 el experimento falla, se ve que hay un límite en términos de bytes enviados para el protocolo especificado:

[Security considerations in gRPC for ASP.NET Core | Microsoft Learn](#)

Message size limits

Incoming messages to gRPC clients and services are loaded into memory. Message size limits are a mechanism to help prevent gRPC from consuming excessive resources.

gRPC uses per-message size limits to manage incoming and outgoing messages. By default, gRPC limits incoming messages to 4 MB. There is no limit on outgoing messages.

On the server, gRPC message limits can be configured for all services in an app with `AddGrpc`:

```
C# Copy

public void ConfigureServices(IServiceCollection services)
{
    services.AddGrpc(options =>
    {
        options.MaxReceiveMessageSize = 1 * 1024 * 1024; // 1 MB
        options.MaxSendMessageSize = 1 * 1024 * 1024; // 1 MB
    });
}
```

Limits can also be configured for an individual service using `AddServiceOptions<TService>`. For more information on configuring message size limits, see [gRPC configuration](#).

En comparación con los tiempos de la práctica 1, tomando cualquiera de los experimentos, vemos una gran diferencia en términos de tiempos mínimos y máximos. Aun tomando la versión con verificación (dónde se recorrían todos los datos para chequear) vemos que los tiempos para un mayor arreglo de datos no llega a 0.003 segundos, mientras en en grpc hay un promedio cercano a 0.14 seg. entre los tiempos mínimos.

Claramente el lenguaje C se destaca en términos de velocidad. Sin embargo en esta práctica con grpc en la desviación estándar hay poca diferencia en relación al crecimiento del tamaño del buffer, mientras que en la práctica 1 esta relación crece exponencialmente. Entonces se puede decir que grpc gana la mano en este aspecto, aunque no se pudieron hacer pruebas con tamaños más grandes de buffer. La estandarización de grpc genera cierta confianza en la solución, considerando su solidez. La desviación estándar es cercana a nula en 10^4 en grpc.

Tiempos practica 1:

PROMEDIO y DESVIACIÓN STD - SHA512

